Laboratorio di Reti - Corso A

WORTH: WORkTogetHer

Progetto di Fine Corso A.A. 2020/21

UNIVERSITÀ DI PISA

Francesco Amodeo

Matricola: 560628

Docente: Laura Ricci

1 Introduzione

Il progetto WORTH consiste nell'implementazione di uno strumento per la gestione di progetti collaborativi che si ispira ad alcuni principi della metodologia Kanban. Quest'ultimo e' un metodo di gestione "agile" di un insieme di attivita' che permette di monitorare la loro evoluzione attraverso una vista sulle fasi del progetto.

Il servizio pertanto deve fornire funzionalita' per la creazione e gestione di progetti formati da una serie di attivita', che chiameremo "card", da portare a termine. Inoltre ogni progetto dispone di una chat utilizzabile solo dai membri dello stesso.

La specifica richiede l'implementazione del servizio secondo un'architettura client-server, con componenti che comunicano utilizzando le diverse tecnologie viste durante il corso, come TCP, UDP e RMI.

Lo stato del sistema deve essere persistente, le informazioni di registrazione e i progetti con le relative card e i relativi membri devono persistere sul file system.

2 Scelte di progetto

2.1 ClientViewController

L'architettura del sistema segue una struttura client-server. L'utente puo' interagire con il servizio Worth attraverso una Command Line Interface, gestita dal componente ClientViewController che riceve ed elabora le richieste dell'utente. Questa componente fa da tramite tra l'implementazione del client e l'utente, invocando le operazioni offerte in base alle richieste ricevute.

2.2 Comunicazione client-server: TCP e protocollo messaggi

Per la maggior parte delle operazioni il client comunica con il server tramite TCP, scambiando messaggi attraverso socket channel. Il messaggio che viaggia tra client e server e' un oggetto Message che modella un messaggio specifico per l'applicazione, include infatti campi per indicare il tipo della richiesta lato client e il tipo di responso lato server. Inoltre permette di includere nel messaggio anche oggetti serializzabili da scambiare per visualizzare e stampare lo stato del sistema (ad esempio progetti, card, ecc..) . Data la eterogeneita' dei messaggi, che a volte possono includere semplici richieste e risposte, ma altre volte anche oggetti, viene utilizzato il meccanismo delle gathering write in modo da scrivere sul canale non un unico ByteBuffer, ma una sequenza.

In particolare sia il server che il client quando scrivono su un canale, scrivono due buffer con un'unica invocazione della write. Nel primo verra' inserito solo un intero che indica il

valore della dimensione del messaggio, nel secondo invece verra' inserito il messaggio vero e proprio. In questo modo il ricevente effettua due read consecutive che gli permettono di allocare il buffer, che andra' a contenere il messaggio ricevuto, della dimensione esatta.

2.3 Server

Il server gestisce le richieste dei client tramite un selettore che seleziona per il server i canali pronti per operazioni di connessione, lettura e scrittura. In particolare quando un canale e' pronto per la lettura, il server effettua le read come descritto sopra e delega l'elaborazione della richiesta ad un task che verra' sottomesso ad un pool di thread. Il task deserializza l'oggetto Message e risolve la richiesta, invocando i metodi della classe WorthImpl, la quale modella il servizio e tutte le sue funzionalita'. Il valore restituito dal metodo di WorthImpl sara' un oggetto Message che conterra' il responso. Il task a questo punto inserisce nell'interestOps() della chiave l'operazione di interesse di scrittura.

2.4 RMI e Callbacks

Per le restanti operazioni vengono utilizzate altre tecnologie. Nel caso della registrazione dell'utente al servizio WORTH, viene utilizzato il meccanismo RMI che permette l'invocazione di metodi da remoto, definiti su interfacce remote. In particolare la classe ServerImpl che modella il server, implementa l'interfaccia remota Server che e' quella che include i metodi invocabili da remoto. Il client tramite questa tecnologia puo' invocare i metodi remoti sullo stub recuperato dal servizio di registry creato nella fase di avvio del server.

Per quanto riguarda le operazioni listUsers() e listOnlineUsers() vengono usate delle callbacks di aggiornamento. Questi metodi mostrano la lista degli utenti registrati al servizio e la lista degli utenti online. Le liste non vengono richieste al server, ma sono salvate localmente lato client e vengono aggiornate tramite le callbacks. Una callback viene innescata in seguito ad un cambiamento di stato della lista degli utenti registrati.

Similmente a quanto visto per il metodo register(), il server invoca il metodo remoto sullo stub del client, fornito come parametro nel momento della registrazione per le callbacks, e aggiorna la lista degli utenti locale al client.

2.5 Chats

Sempre con il meccanismo delle callbacks aggiorna la lista di chat di cui fa parte l'utente in seguito ad un cambiamento di stato della lista dei progetti.

Ogni utente quindi e' interessato a ricevere i messaggi di ogni chat di ogni progetto di cui fa parte. E' richiesto dalla specifica che le chat siano implementate come dei gruppi

multicast. Pertanto il client utilizza un task ChatSniffer che “sniffa” i messaggi inviati su quella chat e li memorizza in una struttura dati, per una lettura futura a partire dall’ultimo messaggio letto.

3 Descrizione classi

Dettaglio delle classi, della loro organizzazione nei package, e sintesi delle funzionalita’ generali.

- **package com.fram3.worth**:
    - **Worth** : interfaccia del servizio che include le operazioni offerte ed i tipi di richieste e responsi possibili
    - **WorthImpl** : implementazione dell’interfaccia Worth e la logica delle funzionalità del servizio WORTH
    - **User** : modella l’utente che interagisce con il servizio
    - **Project :** modella un progetto del servizio
    - **Chat** : modella la chat di un progetto del servizio
    - **Card** : modella una card del servizio
- **package com.fram3.worth.server**:
    - **Server :** interfaccia remota usata dal client per la registrazione al servizio e alle callbacks tramite RMI sull’istanza del server esportata
    - **ServerImpl** : implementa l’interfaccia remota Server e modella la logica del server per il servizio WORTH
    - **RequestHandler** : modella il task che elabora le richieste dei client incaricate dal server
    - **PersistenceManager** : modella il gestore della persistenza dei dati degli utenti e dei progetti del servizio
    - **ServerMain** : contiene il main da cui far partire il server
- **package com.fram3.worth.client:**
    - **Client :** interfaccia remota usata dal server per effettuare le callbacks. Tramite RMI sull’istanza del client esportata, il server notifica il client in seguito a cambiamenti di stato riguardanti utenti e progetti
    - **ClientImpl :** implementa l’interfaccia remota Client e modella la logica del client del servizio WORTH

- **ClientViewController** : modella l'interazione con l'utente. Riceve i comandi e sulla base di questi invoca le operazioni fornite dal client, infine mostra il risultato all'utente
- **ClientMain** : contiene il main da cui far partire il client

- **package com.fram3.worth.utils**:
  - **SecurePassword** : modella l'hashing di una password per la persistenza nel server e le operazioni per futuri confronti tra password fornita e salvata
  - **Message** : modella i messaggi scambiati tra client e server
  - **ChatSniffer** : modella il task che sniffa i messaggi inviati sulla chat, salvandoli nella lista dei messaggi della Chat per consultarli in seguito

4 Threads e strutture dati

4.1 Client

Il client e' cosi strutturato:

- **Thread principale:** manda e riceve messaggi TCP dal server, gestendo le richieste e l'interazione con l'utente
- **Threads sniffers:** tanti thread sniffers quante sono le chat di cui fa parte l'utente. Fanno receive bloccanti sui multicast socket delle chat
- **Thread RMI:** thread che si attiva subito dopo l'esportazione dello stub del client e sta in attesa di invocazioni sui metodi remoti (per le callbacks)

Strutture dati principali: il client mantiene il riferimento all'oggetto User che contiene al suo interno le liste degli utenti e delle chat aggiornate dalle callbacks, la lista di sniffers delle chat dell'utente, e altri campi per la connessione con il server

4.2 Server

Il server e' cosi' strutturato:

- **Thread principale:** accetta le connessioni dei nuovi client, legge e scrive sui canali selezionati dal selettore. Nel caso della lettura dal canale delega l'elaborazione della richiesta al pool
- **Fixed thread pool(10):** riceve ed esegue i task per l'elaborazione delle richieste

- **Thread RMI:** thread che si attiva subito dopo l'esportazione dello stub del server e sta in attesa di invocazioni sui metodi remoti

Strutture dati principali: il server mantiene il riferimento all'oggetto WorthImpl che modella il servizio. Quest'ultimo al suo interno ha le strutture dati principali del servizio: la lista degli utenti registrati e la lista dei progetti creati. Inoltre il server mantiene la lista degli stub dei client iscritti alle callbacks, e il threadpool usato per l'elaborazione delle richieste

4.3 Concorrenza

Per garantire l'atomicita' delle operazioni, ed evitare l'inconsistenza dei dati sono state racchiuse le parti del codice critiche in blocchi synchronized dove si acquisisce la lock sulla struttura dati che viene modificata in quel blocco. In particolare questi accorgimenti si possono vedere nella classe WorthImpl che implementa tutte le operazioni sulle due strutture dati fondamentali del servizio: lista utenti registrati e lista progetti creati. Questi metodi possono essere invocati contemporaneamente da thread RequestHandler diversi e cio' puo' portare ad accessi non atomici delle strutture sopradette e inconsistenza.

5 Librerie utilizzate

L'unica libreria esterna utilizzata e' Gson in versione 2.8.6 , che permette facilmente di trasformare oggetti Java in Json e viceversa. In particolare e' utilizzata per serializzare e deserializzare i dati del sistema in modo da mantenere la persistenza nei file .json degli utenti registrati, dei membri e delle card di progetto. Inoltre e' utile nella comunicazione TCP dove client e server si scambiano i messaggi di tipo Message serializzandoli e deserializzandoli per poi scriverli e leggerli sui canali.

Con tool come maven si puo' aggiungere semplicemente la dipendenza al pom.xml come si vede di seguito:

```
<dependencies>
     <dependency>
         <groupId>com.google.code.gson</groupId>
         <artifactId>gson</artifactId>
         <version>2.8.6</version>
     </dependency>
 </dependencies>
```

altrimenti nella root del progetto si puo' trovare il jar da potere inserire nel classpath.

6 Istruzioni per compilazione e avvio

Il progetto e' stato sviluppato usando maven-3.6.3, ottenendo cosi' la gestione automatica delle dipendenze e un processo di build automatizzato. Per avviare il processo di build basta posizionarsi nella root del progetto ed eseguire il seguente comando:

```
mvn clean package
```

in questo modo viene fatta la clean della target folder che contiene i file risultanti di eventuali vecchie compilazioni e viene ricompilato e testato tutto il progetto, creando l'artefatto .jar . Grazie al plugin maven-assembly-plugin configurato nel file pom.xml, dalla build risulteranno due jar-with-dependencies, uno per il server, uno per il client, contenenti entrambi le librerie esterne utilizzate e con i manifest configurati per eseguire la rispettiva mainClass.

Pertanto per eseguire il server:

```
java -jar target/worthserver-jar-with-dependencies.jar
```

e per il client:

```
java -jar target/worthclient-jar-with-dependencies.jar
```

Alternativamente si puo' compilare ed eseguire direttamente usando i seguenti comandi:

```
javac -d target/classes -sourcepath src/main/java -cp lib/gson-2.8.6.jar src/main/java/com/fram3/worth/**/*.java
```

```
java -cp lib/gson-2.8.6.jar:target/classes com.fram3.worth.server.ServerMain
```

```
java -cp lib/gson-2.8.6.jar:target/classes com.fram3.worth.client.ClientMain
```